2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Daejeon Convention Center
October 9-14, 2016, Daejeon, Korea

RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks

Sebastian G. Brunner*[1], Franz Steinmetz*[1], Rico Belder[1] and Andreas Dömel[1]

***Abstract*— Robotic tasks are becoming increasingly complex, and with this also the robotic systems. This requires new tools to manage this complexity and to orchestrate the systems to fulfill demanding autonomous tasks. For this purpose, we developed a new graphical tool targeting at the creation and execution of robotic tasks, called RAFCON. These tasks are described in hierarchical state machines supporting concurrency. A formal notation of this concept is given. The tool provides many debugging mechanisms and a GUI with a graphical editor, allowing for intuitive visual programming and fast prototyping. The application of RAFCON for an autonomous mobile robot in the SpaceBotCamp competition has already proved to be successful.**

I. Introduction

Robotic systems are growing more and more complex. Usually, they consist of a heterogeneous composition of different modules for specific purposes (manipulation, vision, navigation, etc.). All these components need to be coordinated both in terms of communication and task flow. ROS [1] is a common system handling the communication. Yet in this paper, we focus on managing the task flow (over time), which defines the flow control and thus orchestrates all modules of a robotic system.

One approach for specifying tasks on a high level are symbolic task planners, such as CRAM [2]. On the one hand, their reasoning system and access to world knowledge databases allow for complex tasks often found in household scenarios (e. g. setting a table). On the other hand, these tools haven't reached a level in which they can be applied for specific real-world tasks, especially in industries. This is partially due to their obligatory (world) model, which is often either under or over constrained and fails if a failure during the execution cannot be represented in it [3]. Moreover, the demand for computational resources is significantly higher.

As an alternative, many robotic systems use approaches based on (hierarchical) state machines. They are a common way of defining what to do in which order and how to react to certain environmental changes or events. Here, the system is typically in one state, from which it can proceed to a defined number of other states. Which one is chosen at run-time, depending on external events (like sensor inputs) or internal events (like the result of a successful state execution).

Originally, state machines have been programmed textually as code, which does not only require expertise in programming but also results in huge code fragments for non-trivial tasks. Such code is hard to maintain, debug and

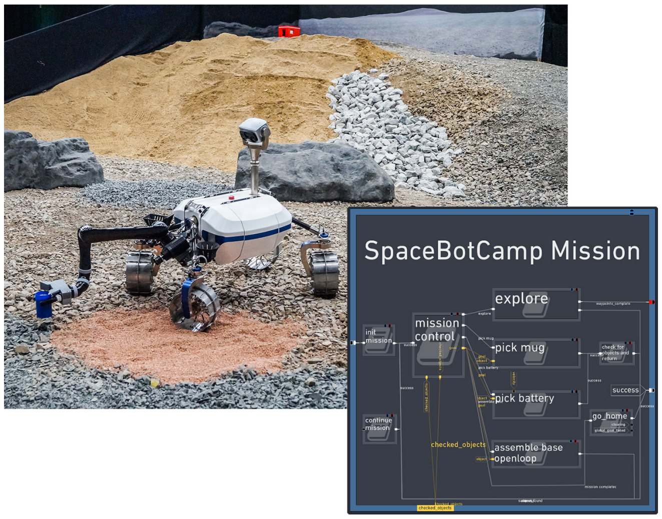


Fig. 1. Our LRU robot [4] at the SpaceBotCamp 2015: Using a RAFCON state machine (lower-right corner), the robot automatically grasps the blue container for rock sample before returning it to the red base in the background for assembly.

understand. Besides, many programming languages (such as C[++]) are compiled. This leads to tedious development cycles, as for every change in the program, the compiler must be run, the binaries deployed and the system restarted.

In this paper, we present a newly developed tool called RAFCON (*RMC advanced flow control*). It allows for visual programming of hierarchical state machines, using a feature-rich graphical user interface (GUI). It is programmed in the interpreted language Python, which relieves one from compiling and makes it possible to alter a state machine, even while it is running. Python enables the integration of heterogeneous components of a robotic system written in different languages. The novel visualization of hierarchical state machines together with sophisticated debugging mechanisms permit fast prototyping and intuitive task development with closed loop system tests, without the need of deep programming skills. The architecture enables the collaboration in a team on a single state machine. We proved this in the SpacebotCamp 2016[1], in which RAFCON was used for the mission control (see Fig. 1). The task of this contest included autonomous exploration in an unstructured terrain, as well as the localization, collection and assembly of different objects.

The next sections are organized as follows: We first put the work in context and give related work (Section II). Then, the core concepts of a RAFCON state machine are explained

*Both authors contributed equally to this work.
[1]All authors are with Robotics and Mechatronics Center (RMC) of the German Aerospace Center (DLR), Oberpfaffenhofen-Wessling, Germany. `firstname.lastname@dlr.de`

[1]http://s.dlr.de/ura7

(Section III). A formal definition of the state machine is given in Section IV. The next Section V presents the GUI working on top of the state machine core. More features of RAFCON are listed in Section VI. An experimental validation is shown in Section VII. Finally, Section VIII concludes the paper and gives an outlook on future work.

II. Related work

RAFCON falls in a broad category, which can be named *robot programming*. This field is huge and includes *Programming by Demonstration* utilizing *teleoperation*, *kinesthetic teaching* or *observational learning* [5]. Using these methods, a robot can be taught arbitrary motions. However, definitions how to react on certain inputs, like camera images, or external events are very limited.

Another approach are the aforementioned *symbolic task planners*, for which CRAM [2] is an example. In theory, those systems accept natural high-level commands such as "set the table for breakfast“ and then plan the involved steps using a reasoning and prediction system with the help of detailed models and world knowledge. This is a top-down approach, contrasting the bottom-up approach of hierarchical state machines.

Finite state machines (FSM, or extensions of them) are heavily used for programming complex software or representing extensive robot behaviors [6]–[8]. Complex decision trees would otherwise be very tedious to describe in huge switch and if-else cascades ("spaghetti code"). A well known kind of state machine is the *deterministic finite state machine* (DFA) [9]. As state machines do not produce any output, the *finite state transducer* (FST) augments the DFA with an output alphabet and an output function [9].

Another extension of FSM is *statechart*, originally invented by Harel [10]. Some of the concepts introduced by Harel are orthogonality (which is equal to concurrency), hierarchy and a special form of broadcast communication.

A further statechart dialect is *SyncCharts* [11]. It also includes concurrency, hierarchy and information exchange concepts, yet in addition offering enhanced preemption capabilities and support for transition priorities. The clear focus of SyncCharts lies on *reactive systems*, i. e. systems reacting to external events. The fact that there exists an exact formal notation for SyncCharts is a clear advantage over statecharts.

Flowcharts are another type of diagram for modeling behaviors or processes. The graphical representation of state diagrams like statechart and SyncCharts share some similarities with flowcharts. Yet, the formalisms heavily differ. One distinction is the way in which a transitions is determined. State diagrams are event-driven, i. e. events trigger their corresponding transitions. For flowcharts, transitions are either automatically chosen (if there is only one) or depend on the outcome of a state [12]. Strictly speaking, flowcharts are not state machines, however we include these charts in our broader sense of state machines.

For RAFCON, we deliberatively decided against an event-driven design. State diagrams may have advantages in reactive systems in which events are in the foreground, such as stopwatches [10] or watchdogs for sensor monitoring [11]. However, with increasing complexity, event-driven system are error-prone [13], resulting from complex topics that need to be tackled, like event caching and priorization, event expiration and parallel event handling [14]. Therefore, RAFCON is inspired more by flowcharts, letting states decide about the next transition to follow. Nevertheless, this decision can be based on events, if the state registered a callback.

In our opinion, statecharts and SyncCharts offer too many concepts, which increase complexity and whose benefit is arguable. For example, they allow transitions to cross hierarchy boundaries, which is contrary to important software engineering principles like encapsulation and composability [15]. There are many more concepts (transition guards, transition hooks, deferred events, ...). All of these can unburden the programmer from some work, while at the same time increasing complexity and with this error-proneness of the underlying system. Therefore, the RAFCON core (see Section III) is kept slim, with only an essential feature set.

All state machine types mentioned so far are general concepts that are not bound to robotic applications. There exist also many solutions clearly invented for robotic purposes. In [16], robots are programmed using generic action components which are mapped to statecharts. Another state machine like system for representing robot tasks are skill nets [17]. However, their range of application is rather limited, as they lack concepts for hierarchy, concurrency and explicit data handling. XABSL [18] uses hierarchical state machines to define robotic behaviors.

A proper tool for programming complex robotic tasks must not only have concepts for hierarchies and concurrencies but also needs a GUI intuitively integrating these concepts and allowing for efficient programming. There is a huge number of visual programming tools for a variety of different domains[2]. Worth mentioning is for example *Scratch*, which is intended for education and allows for both imperative and event-driven programming [19]. The idea of using a visual state machine concept for programming robots is not new, either. One famous example for programming LEGO Mindstorms robots is *NXT-G* [20]. There are also a lot of hardware-independent tools, however all of them are restricted in some way. Some are no longer maintained, like *ROS Commander* [8], *RobotFlow* [21] or *MissionLab* [22], others are closed source/commercial, such as *Gostai Studio* [23], or lack a graphical editor, for example *SMACH* [3].

Further state machine libraries that do not have a graphical user interface (GUI) at all also exist, such as *Boost statechart* [24] or *Spring Statemachine* [25].

While all of the mentioned tools were not considered as sufficient for our needs, we adopted many or their features for RAFCON. SMACH for example has a similar concept of outcomes and data ports, called input/output keys. NXT-G visualizes its input and output data ports as well as the data flows, just as RAFCON. ROS Commander includes a

[2]See http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/ for a graphical overview

graphical editor, however it only offers a limited number of features.

In our opinion, a GUI with a graphical representation of the state machine and the option of editing, is an important supportive feature of a programming tool for robots. However, this is not an easy feature to achieve, as bigger state machine with deep hierarchies tend to become very complex. Most visual programming tools do not support hierarchies, like NXT-G, or show elements on different hierarchy levels in the same size, such as SMACH, which does not scale.

The graphical representation of RAFCON is inspired by the flow control tool *Bubbles*, which has been developed at our institute some years ago [26].

III. Core framework

The heart of the RAFCON core framework is a hierarchical state machine whose components are shown in the class diagram in Fig. 2 and in their usage in Fig. 4. These components are further explained in the following section.

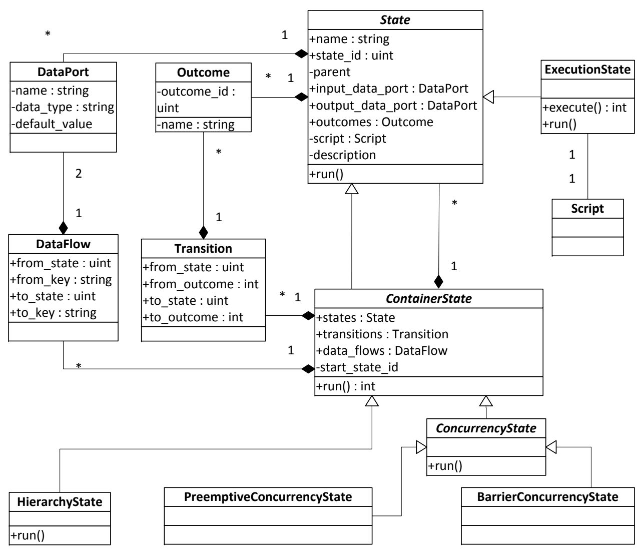


Fig. 2. A class diagram of the basic state machine classes of RAFCON.

A. States

Every state machine must have a single state at its root, the so called *root state*. There are three basic types of states: *ExecutionState*s, *HierarchyState*s and two forms of *ConcurrencyState*s.

When thinking of a hierarchical state machine as a tree, then its leaves are typically the ExecutionStates, see Fig. 3. These states cannot contain any child state but instead each of them contains a Python script to be run when the state is executed. This allows the state to carry out arbitrary functions, like calling an external robot API or triggering a robot middle-ware such as ROS.

HierarchyStates are *ContainerStates* for child states of any type and are thus called parent states of their child states. The purpose of HierarchyStates is simply to allow for the definition of hierarchical structures by grouping related states. The direct children of a HierarchyState can be considered as a flat state machine on its own with a single entry state.

ConcurrencyStates are also ContainerStates. However, all its child states are executed in parallel. There are two subtypes, *PreemptiveConcurrencyState*s and *BarrierConcurrecyState*s. If a child state of a PreemptiveConcurrencyState finishes, all its sibling states (i.e. states that have the same parent state) are preempted. In contrast, the BarrierConcurrecyState waits for all of its child states to finish before continuing. ConcurrencyStates are crucial for applications in which several tasks have to be accomplished at the same time. One example is running one or more observer states monitoring the environment with different perception devices, while at the same time an object is manipulated.

LibraryStates are encapsulated state machines, which can be used and treated as any other state. This allows for easy reusability and eases the collaboration on bigger state machines.

We emphasize that a state in a RAFCON state machine does not correspond to a state of the environment (world state) but to a state in the flow of a robotic task.

B. Outcomes and transitions

Each state has outcomes that define connection points for transitions. The aforementioned scripts of ExecutionStates return a value corresponding to one of the state's outcomes. The successor state is determined by following the transition connected to the returned outcome. If its target is a sibling state, then this state is executed consequently. Alternatively, the transition can be connected to an outcome of the parental ContainerState. In this case, the execution of the ContainerState is left on the outcome the transition is connected to.

If a HierarchyState is to be executed, the *start transition* determines which of its child states to execute first. For ConcurrencyStates, no start transitions are required, as all inner states are executed in parallel.

Another core functionality of the RAFCON framework is its integrated error and preemption handling (similar to SMACH [3]). Every state must have the two outcomes *aborted* and *preempted*. If an uncaught exception occurs in a state, it is automatically left on the aborted outcome. In PreemptiveConcurrencyStates, child states being preempted due to a sibling state having finished its execution are automatically left on the preempted outcome. Based on this, a programmer can implement arbitrary error recovery strategies or shutdown mechanisms in states connected to the appropriate outcomes. In case a triggered aborted/preempted outcome has no transition connected to it, then the aborted/preempted outcome of the parent state is triggered. This propagation is continued until an outcome with a transition is found. Thereby, states handling preemption and errors can be placed at any level. PreemptiveConcurrencyStates having caused a preemption stop its propagation. In contrast, errors propagate up to the root state and stop the execution, if the error is not caught.

C. Data ports and data flows

RAFCON supports concepts for private, scoped and global variables. Private variables can be defined within the script of an ExecutionState and do not leave this scope. However, states can pass values to other states using data ports and data flows, which relates to scoped variables. For this, every state can define input data ports that correspond to parameters of a function. Similarly, a state defines output data ports, corresponding to return values of functions. Each data port has a name, data type and default value. Values on input data ports are passed to the script of an ExecutionState. This script can then assign values to the output data ports of its state. Finally, variables globally accessible are stored in the global variable manager. This manager supports storing variables by value and by reference and features thread safe read and write accesses.

Data flows are directed and define the mapping between data ports. If a data flow connects two data ports, then the value assigned to its source is forwarded to its target. By this, values cannot only be passed between sibling states, but also up and down in the hierarchy.

IV. Formal state machine description

The concepts of a RAFCON state machine, described in the previous section, contain components distinguishing it from all other state machines mentioned in Section II. Therefore, in the following, we will give a formal notation of our state machine using predicate logic and set theory.

Concerning the execution semantics, the RAFCON state machine model resembles a FST with the Moore model [9], as the actions of the state machine take place during the execution of a state and not during the transition.

However, relating to data handling, our state machine approach is similar to the Mealy model [9]. The main difference to the Mealy model is that the input for a state is not associated with the transitions but is passed via dedicated data edges, called data flows. More specifically, states forward their output as input to other states (1×n relation).

First, we define the elements of a state machine and its relationships. Then, the definition of a RAFCON state machine is provided.

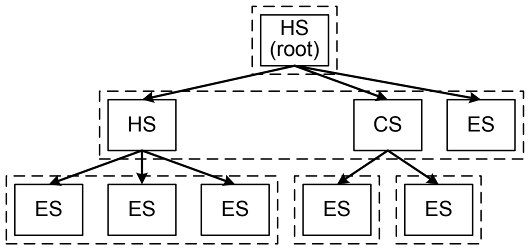


Fig. 3. The hierarchical state machines in RAFCON can be represented as a tree. The root state, all direct children of a hierarchy state (HS), as well as each direct child of a concurrency state (CS) can be considered as a separate, flat state machine (dashed rectangles). The leaves of the tree are typically execution states (ES), but could also be empty hierarchy or concurrency states.

Definition 1 (State): A state s is an entity having an execution mode and type: $\mathrm{sem}(s) \in \{\text{active, inactive}\}$, $\mathrm{type}(s) \in \{\text{execution, hierarchy, concurrency}\}$. The different states are called *execution states*, *hierarchy states* and *concurrency states*, respectively.

Definition 2 (RAFCON state machine tree (RST)): A RAFCON state machine tree (RST) is a *rooted tree* $G = (S, E)$ with $r \in S$ being the *root vertex*. S is the set of nodes and E is the set of edges which define the *tree-order* $x < y$ $(\forall x \in V \setminus r : r < x)$, following the definition of [27, p. 13–15]. All vertices $s \in S$ are states, therefore r is also titled the *root state*. All states $s \in S$ with $\mathrm{type}(s) = \text{execution}$ must be *leaves* of the tree. In Fig. 3, an example of a RAFCON state machine tree is shown.

Using the tree-order of a RST, we can define further relations on such a tree:

Definition 3 (parent): In a RST $G = (S, E)$, state $p \in S$ is called a parent of state $s \in S$, if $p < s$ and p and s being *neighbors* [27, p. 3] (i.e. share a common edge $e \in E$). This is written as $\mathrm{parent}(p, s)$.

Definition 4 (children): In a RST $G = (S, E)$, the children of a state $p \in S$, written as $\mathrm{children}(p)$, is defined as the set of all $s \in S : \mathrm{parent}(p, s)$.

Definition 5 (siblings): In a RST $G = (S, E)$, two states $s, s' \in S$ are said to be siblings, write $\mathrm{siblings}(s, s')$, if $\exists p \in S : \mathrm{parent}(p, s) \wedge \mathrm{parent}(p, s')$. Note that s and s' can be the same state.

The following definitions 6 and 7 are related to the logical flow of a RAFCON state machine.

Definition 6 (Logical port): A logical port is a couple $l = (s, t)$ with state s and type $t \in \{\text{income, outcome}\}$, $\text{outcome} \in \{\text{aborted, preempted, common}\}$. A logical port with any type of outcome is literally called an *outcome*.

Definition 7 (Transition): A transition is a couple $t = (l_1, l_2)$ with $l_1 = (s_1, \text{outcome})$, $l_2 = (s_2, \text{income})$ both being logical ports and $\mathrm{siblings}(s_1, s_2)$. l_1 is called the *source port*, l_2 the *target port*.

The following definitions 8 through 11 are related to the data handling of a RAFCON state machine.

Definition 8 (Data port): A data port is a triple $d = (s, t, v)$ with state s, type $t \in \{\text{input, output}\}$ and default value $v \in A$, whereat A is an arbitrary alphabet.

Definition 9 (Data flow): A data flow is a couple $F = (d_1, d_2)$ with $d_1 = (s_1, t_1, v_1)$ and $d_2 = (s_2, t_2, v_2)$ both being data ports, which must fulfill

$$\begin{aligned}&(\mathrm{siblings}(s_1, s_2) \wedge t_1 = \text{output} \wedge t_2 = \text{input}) && \vee\\ &(\mathrm{parent}(s_1, s_2) \wedge t_1 = t_2 = \text{input}) && \vee\\ &(\mathrm{parent}(s_2, s_1) \wedge t_1 = t_2 = \text{output})\end{aligned}$$

Thus, a data flow exists either between two sibling states or between a child and its parent state.

Definition 10 (Data port value): Each data port $d = (s, t, v)$ has a value a of a certain alphabet A: $\mathrm{value}(d) = a$. By default, $\mathrm{value}(d) = v$. If $t = \text{output}$ and $\mathrm{type}(s) = \text{execution}$, then the output function μ_e can determine this value (see Definition 12). If there is a data flow $f = (d', d)$, then the value of the source port d' defines the value of the target port d: $(\mathrm{value}(d') = a') \Rightarrow (\mathrm{value}(d) = a')$.

Definition 11 (Value vector): A value vector $V_{s',t'}$ with state s' and data port type $t' \in \{\text{input}, \text{output}\}$ is a vector consisting of all values assigned to all ports of that type at that state, thus $[\text{value}(d_1), \text{value}(d_2), \ldots]$ for all $d_i = (s = s', t = t', v_i)$.

With the definitions of the elements of a state machine and its relations, the state machine definition can be given:

Definition 12 (HFPD): A *hierarchical, parallel, finite state machine with data flows* (HPFD) is a 11-tuple $(\Sigma, W, G, L, T, D, F, M, S_{\text{exit}}, T_{\text{exit}}, P)$, where

- Σ is the data alphabet. This is typically $\mathbb{R}$, but can also contain e. g. strings or lists.
- $W(\tau) : \tau \mapsto \Sigma^n, n \in \mathbb{N}$ is a vector function representing the world state at time instance τ. This vector function serves as access to global variables, see Section III, or sensor readings. In robotic contexts, Σ is not finite and thus neither are the elements of $W(\tau)$.
- $G = (S, E)$ is a RST, with the set of states S being finite and non-empty.
- L is the set of logical ports, whereat $\forall s \in S : (\exists! l \in L : l = (s, \text{income})) \land (\exists! l' \in L : l' = (s, \text{aborted})) \land (\exists! l'' : l'' = (s, \text{preempted}))$. Thus, all states must have exactly one logical of each type income, aborted and preempted. Furthermore, $\forall l = (s, t) \in L : s \in S$.
- T is the set of transitions, with $\forall t = (l_1, l_2) \in T : l_1 \in L \land l_2 \in L$.
- D is the set of data ports, with $\forall d = (s, t, v) \in D : s \in S \land v \in \Sigma$.
- F is the set of data flows, with $\forall f = (d_1, d_2) \in F : d_1, d_2 \in D$.
- M is the set of all output functions $\mu_e : V_{e,\text{input}} \times W(\tau) \mapsto L_{e,\text{outcome}} \times V_{e,\text{output}}$ for the execution states $e \in S$. Here, $V_{e,\text{input}}$ and $V_{e,\text{output}}$ are the input respectively output value vectors of e and $L_{e,\text{outcome}}$ being the set of all logical ports $l = (s = e, t = \text{outcome}) \in L$. The output functions have access to the current world state $W(\tau)$. Together with the input port values, they determine the outcome port and the values assigned to the output ports. The evaluation of an output function takes a certain amount of time $\Delta\tau > 0$.
- $S_{\text{entry}} \subset S$ is the set of all *start states*. A start state is the state evaluated first when entering a hierarchy state. Thus, for each hierarchy state $h \in S$, there is at most one state $s \in S_{\text{entry}}$ with $\text{parent}(h, s)$.
- T_{exit} is a set of couples $t_{\text{exit}} = (l_1, l_2)$ with $l_1 = (s_1, t_1 = \text{outcome}) \in L$ being the source outcome, $l_2 = (s_2, t_2 = \text{outcome}) \in L$ being the target outcome and $\text{parent}(s_2, s_1)$. If the execution (see further down) reaches an outcome being the source port of a couple $t_{\text{exit}} \in T_{\text{exit}}$, then the state s_2 of the target outcome is left at this target outcome l_2.
- P is the set of all outcome functions $\rho_c : \mathbb{P}(L_{\text{children}(c),\text{ outcome}}) \mapsto L_{c,\text{outcome}}$ for the concurrency states $c \in S$. Here, $\mathbb{P}(L_{\text{children}(c),\text{ outcome}})$ is the power set of all outcomes of all $\text{children}(c)$ and $L_{c,\text{outcome}}$ is the set of all logical ports $l = (s = c, t = \text{outcome}) \in L$. If one child state of c exits (c being a *preemptive concurrency state*) or all children of c have exited (c being a *barrier concurrency state*), then ρ_c determines the outcome of c on which c is exited.

The following properties apply for HPFDs:

- If a hierarchy state is executed, then the execution directly starts with the entry state defined by S_{entry} and finishes when an exit outcome defined by T_{exit} is reached.
- If a concurrency state is entered, all of its children are entered in parallel. Thus, the execution splits up. The outcome function ρ_c defines on which outcome to leave the concurrency state.
- Initially, all states are inactive: $\forall s \in S$, $\text{sem}(s) = \text{inactive}$. When a state s is entered, meaning that the execution reaches an income $l = (s, \text{income}) \in L$, then $\text{sem}(s) = \text{active}$. Accordingly, when a state is left, meaning that the execution reaches an outcome $l = (s, \text{outcome}) \in L$, then $\text{sem}(s) = \text{inactive}$.
- Every port $l = (s, t = \text{outcome}) \in L$ can only be the source of either one $t \in T$ or one $t_{\text{exit}} \in T_{\text{exit}}$.
- Transitions are only allowed in hierarchy states.

With these definitions, for example deadlock detection and reachability analysis are possible in future work. Furthermore, it enables comparisons with other state machine approaches.

V. Graphical user interface

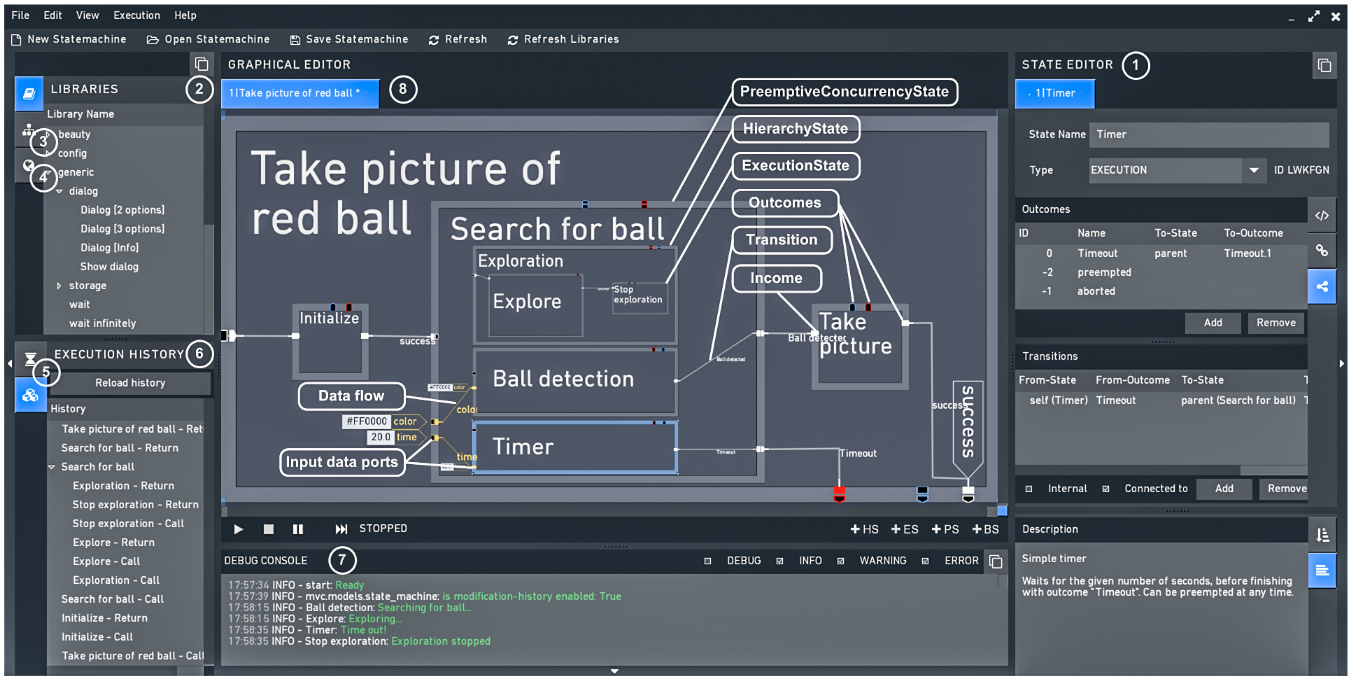
The GUI is an important feature of RAFCON. A screenshot of it with an example state machine is shown in Fig. 5. The user interface distinguishes RAFCON from other flow-control tools, especially with its central element, the graphical state machine editor. Using this GUI, state machines and their execution status can be viewed and observed. On top of that, state machines can be created from scratch and edited, even during the execution of a state machine.

The GUI allows for visual programming. Compared to textual programming, most of the work can be accomplished straightforward using only the mouse; code is replaced by a compact graphical representation. This helps in creating a mental model of a state machine [28]. Both logical and data connections can be seen on the first sight. In addition, the current execution point(s) are highlighted. Programming state machines visually is fast and getting started is easy, as the interaction is intuitive. This is supported by the layout and design of the GUI, which was developed in cooperation with a professional interface designers[3]. A video showing how to program tasks in RAFCON is provided next to this paper.

A. GUI layout

The GUI is implemented with the GTK+ widget toolkit. It is completely separated from the RAFCON core. Changes made to the core are published using the observer pattern to the model-view-controller architecture (MVC) of the GUI. The layout and design of the GUI follow commonly accepted principles. One example for this is the arrangement

[3]Interaktionswerk, https://interaktionswerk.de/



Library Name
- beauty
- config
- generic
- dialog
  - Dialog [2 options]
  - Dialog [3 options]
  - Dialog [Info]
  - Show dialog
- storage
- wait
- wait infinitely

History
- Take picture of red ball - Ret
- Search for ball - Return
- Search for ball
  - Exploration - Return
  - Stop exploration - Return
  - Stop exploration - Call
  - Explore - Return
  - Explore - Call
  - Exploration - Call
- Search for ball - Call
- Initialize - Return
- Initialize - Call
- Take picture of red ball - Call

State Name: Timer
Type: EXECUTION
ID LWKFGN

Outcomes

ID	Name	To-State	To-Outcome
0	Timeout	parent	Timeout.1
-2	preempted		
-1	aborted		

Transitions

From-State	From-Outcome	To-State
self (Timer)	Timeout	parent (Search for ball)

Internal [✓ Connected to]

Description

Simple timer

Waits for the given number of seconds, before finishing with outcome "Timeout". Can be preempted at any time.

```
17:57:34 INFO - start: Ready
17:57:39 INFO - mvc.models.state_machine: is modification-history enabled: True
17:58:15 INFO - Ball detection: Searching for ball...
17:58:15 INFO - Explore: Exploring...
17:58:35 INFO - Timer: Time out!
17:58:35 INFO - Stop exploration: Exploration stopped
```

Fig. 4. Screenshot of the RAFCON GUI, showing an example of a simple state machine for a robotic task. The robot explores the environment until a red ball is detected or the time runs out.

and alignment of elements following the gestalt principles proximity, similarity, continuity and closure, which improves the effectiveness of information retrieval [29].

Flexibility and modularity are further characteristics of the GUI. The aforementioned graphical editor is the central element of the GUI (see V-B) and typically takes the most space. It is surrounded by three sidebars. The scalable, detachable and foldable sidebars are organized in notebooks, whose tabs can be reordered.

The right sidebar is named the *state editor*, see (1) in Fig. 4. The selected state is shown in here with all its details. Common state properties such as name and description can be changed. In addition, there are several widgets to modify outcomes, input/output data ports, *scoped variables* (a feature to graphically ease connecting input and output ports of sibling states), transitions and data flows. For execution states, a source code editor allows for altering the code with code highlighting and syntax check.

The left sidebar is composed of several widgets with more state machine related content:

(2) *Library manager*: shows all available libraries, which can be added to the current state machine using drag and drop

(3) *State machine tree*: gives another tree perspective on a state machine and allows for navigation

(4) *Global variable manager*: shows the value of all global variables

(5) *Modification history*: allows to undo and redo changes made to the state machine. The modifications history is stored as a tree. The widget allows to access the different branches of the history.

(6) *Execution history*: lists the executed states in chronological order and optionally shows the context data

The sidebar at the bottom only consists of a logger (7). All log entries, from the core, the GUI and the executed states are shown here. The messages are color coded and can be filtered by log level.

B. Graphical editor

A lot of effort has been put into the development of the graphical state machine editor, see (8) in Fig. 4. The biggest challenge here is the proper visualization of huge state machines with deep hierarchies, as mentioned in II. A user should be able to quickly understand the purpose of a state machine and easily navigate both within one hierarchy level and up and down the hierarchy.

For this, a panning and zooming mechanism was introduced. Elements down in the hierarchy are smaller than further up in the hierarchy. Details disappear when they get too small. This intuitive approach, common in digital maps, helps in navigating even complex state machines easily.

Fully zoomed out, the user only compactly sees the highest-level states and their connections. Therefore, it is easy to grasp the coarse task flow, as one is not distracted by the details of the implementations of the single steps within a task. If one is interested in the composition of a state, the mouse cursor can be moved over the state and the scroll wheel turned. Consequently, the state is zoomed in and more details of its inner content (child states, transitions, etc.) are revealed.

Further interaction like moving or resizing states, adding connections with via-points is possible as well. The control of these actions is similar to that of other applications, such as UML editors.

There are different viewing modes implemented, which can be activated and allow to further focus on currently relevant aspects of a state machine. By default, everything is shown. Other view modes hide all data flow related entities or put everything in the background, except the currently selected state with its connections.

If a state is selected, it is marked with a blue border and its details are shown in the states editor. There it can further be inspected and manipulated.

VI. Further features

Not all of the features of RAFCON have been described yet. While it is not possible to mention all in the scope of this paper, at least the most important remaining ones shall be listed here.

RAFCON stores state machines in a human readable file format. Each state is stored in its own folder on the file system, containing separate JSON files for core and GUI data. The folder structure directly reflects the state machine tree. This enables versioning of state machines in repositories and thus collaborative state machine programming.

The comprehensive core API allows for programmatic state machine generation from within another tool, e. g. task planners like CRAM.

Modifications of the state machine can be conducted during runtime. The script code of ExecutionStates can be changed as well as the structure of the state machine.

Using the stepping mode of the execution engine, a state machine can easily be debugged. Similar to the debug mode in modern IDEs, one can step over, into and out. Hereby, values assigned to data ports are shown in the GUI.

On top of that, one can also step backwards, due to the execution history being stored including the whole data context. To the best of our knowledge, this feature is novel in the context of debugging capabilities of IDEs or visual programming tools. For ExecutionStates, instead of the normal script, an optionally defined backwards-execution script is executed. This helps debugging complex state machines as it eases the way to return to a certain execution point.

Finally, the state machine can be started at an arbitrary state, which can be programmatically specified by its unique state path or graphically selected by the user.

During development, the RAFCON GUI is essential for programming state machines (see chapter V). When the state machine is ready for release, the RAFCON core can execute the state machine on the target device without any GUI overhead. This makes our framework very lightweight. Yet, the GUI allows the remote control of state machines running on another host.

VII. Case studies

RAFCON is a software tool which aims at offering an intuitive way to program robots and helps to build a clear hierarchical system architecture. However, it is difficult to give a statistical evaluation of its usability and intuitiveness or quality measurements of systems designed with our tool.

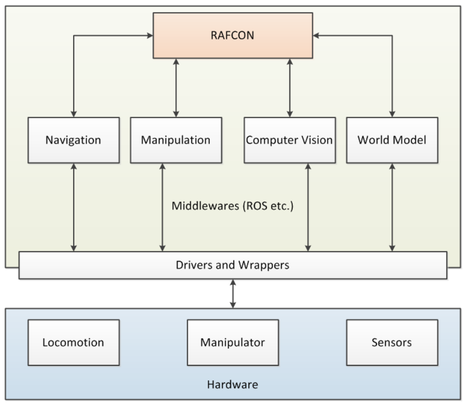


Fig. 5. A coarse overview of the role of RAFCON, orchestrating the different system components in the SpaceBotCamp competition.

Thus, we focus on demonstrating successful applications of our software in challenging scenarios.

In the SpaceBotCamp Competition held in November 2015 and organized by the DLR Space Administration, we participated together with nine other teams. The mission was to explore a previously unknown, moon like environment with a mobile robot (see Fig. 1), find two objects and finally bring them to a base station for assembly. A time limit of sixty minutes had to be met and the communication to the robot was delayed by four seconds round trip time. Thus a very high level of autonomy was necessary [30].

Our team was the only one to fulfil all tasks of the competition specification. Furthermore we accomplished the mission fully autonomously, requiring only half of the given time limit. Next to a robust hardware design and sophisticated solutions for navigation, manipulation and computer vision, our RAFCON state machine concept played a vital role in orchestrating all system components (for the coarse architecture, see Fig. 5).

Our final state machine had a complexity of more than 700 states, more than 1200 transitions and eight hierarchy levels. This is a high number, considering that the states are only performing high level actions (like retrieving some knowledge of a world model or driving the robot autonomously to another viapoint).

RAFCON helped us to apply the divide an conquer concept to the task. This enabled several developers to program the state machine at the same time. Specifically, states concerning navigation, exploration, computer vision, object detection, world model integration, manipulation and grasping, were all created independently. Beyond that, the initial strategy for solving the challenge could be mapped out before implementing the actual functionalities. Moreover, having wrapped all system functionalities into states, altering the behavior of our robot using the GUI was much simpler compared to purely textual based state machine descriptions.

RACELab is another project at our institute, in which advanced robot skills for industrial automation are developed [31]. These skills fully rely on RAFCON and demonstrate its usefulness in industrial contexts with robot arms.

SAPHARI[4] is a further project example, in which the use of RAFCON proved to be fruitful for human-robot-interaction.

VIII. Conclusions

We presented RAFCON, a graphical tool to facilitate collaborative programming of complex robotic tasks. Hierarchical state machines allow the creation of concurrent flow controls. The intuitive GUI and many features like the debug mechanisms of the execution engine extend its usefulness. This was proved especially in the SpaceBotCamp competition, in which our robot controlled by RAFCON fulfilled the challenging mission of autonomous exploration as well as object detection, collection and assembly.

RAFCON is constantly being extended and improved. To increase the usability and overview while facing complex state machines, the GUI is further enhanced to match the design guidelines of our designer. One extension of the GUI will be a miniature view of the state machine, which could serve as a time saving navigation alternative. Another planned feature useful for both multi-robot and mobile robot scenarios is the ability to remotely edit state machines.

We noticed the deficiency of a properly designed, well supported and powerful task programming tool freely available. Therefore, we plan to provide RAFCON as an open source tool to the community[5].

Acknowledgment

This work has been funded by the Helmholtz-Gemeinschaft Germany as part of the project RACELab, by the Helmholtz Association, project alliance ROBEX, under contract number HA-304 and by the European Commission under contract number FP7-ICT-608849-EUROC.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, 2009.

[2] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM-A Cognitive Robot Abstract Machine for everyday manipulation in human environments," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 2010, pp. 1012–1017.

[3] J. Bohren and S. Cousins, "The SMACH high-level executive [ROS news]," *IEEE Robotics & Automation Magazine*, vol. 4, no. 17, pp. 18–20, 2010.

[4] A. Wedler, B. Rebele, J. Reill, M. Suppa, H. Hirschmüller, C. Brand, M. Schuster, B. Vodermayer, H. Gmeiner, A. Maier, B. Willberg, K. Bussmann, F. Wappler, and M. Hellerer, "LRU - Lightweight Rover Unit," in *ASTRA*, 2015.

[5] A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration," in *Springer handbook of robotics*. Springer, 2008, pp. 1371–1394.

[6] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011, pp. 5568–5575.

[7] S. Jentzsch, S. Riedel, S. Denz, and S. Brunner, "TUMsBendingUnits from TU Munich: RoboCup 2012 Logistics League Champion," in *RoboCup 2012: Robot Soccer World Cup XVI*, ser. Lecture Notes in Computer Science, X. Chen, P. Stone, L. Sucar, and T. van der Zant, Eds. Springer Berlin Heidelberg, 2013, vol. 7500, pp. 48–58.

[8] Hai Nguyen, M. Ciocarlie, Kaijen Hsiao, and C. Kemp, "ROS commander (ROSCo): Behavior creation for home robots," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, 2013, pp. 467–474.

[9] K. Krithivasan, "Theory of Automata, Formal Languages and Computation," 2014.

[10] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[11] C. André, "SyncCharts: A visual representation of reactive behaviors," *Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis*, 1995.

[12] A. J. Simons, "On the Compositional Properties of UML Statechart Diagrams." in *Rigorous Object-Oriented Methods*, 2000.

[13] S. Parent, "A Possible Future of Software Development," in *Keynote talk at the Workshop of Library-Centric Software Design*, 2006.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] I. Maier, T. Rompf, and M. Odersky, "Deprecating the observer pattern," Tech. Rep., 2010.

[16] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using UML/P Statecharts," in *Robotics and Automation (ICRA), IEEE International Conference on*, 2013, pp. 461–466.

[17] U. Thomas, B. Finkemeyer, T. Kroger, and F. Wahl, "Error-tolerant execution of complex robot tasks based on skill primitives," in *Robotics and Automation (ICRA), IEEE International Conference on*, vol. 3, 2003, pp. 3069–3075.

[18] M. Loetzsch, M. Risler, and M. Jungel, "XABSL - a pragmatic approach to behavior engineering," in *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, 2006, pp. 5124–5129.

[19] M. Resnick, J. Maloney, A. Monroy Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[20] J. F. Kelly, *Lego Mindstorms NXT-G Programming Guide*. Apress, 2010.

[21] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raïevsky, M. Lemay, and V. Tran, "Code reusability tools for programming mobile robots," in *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, 2004, pp. 1820–1825.

[22] R. Arkin, "Missionlab v 7.0," 2006.

[23] J.-C. Baillie, A. Demaille, Q. Hocquet, M. Nottale, and S. Tardieu, "The Urbi universal platform for robotics," in *First International Workshop on Standards and Common Platform for Robotics*, 2008.

[24] A. H. Dönni, "The boost statechart library," *Internet: http://www.boost.org/doc/libs/1_46_0/libs/statechart/doc/index.html*, p. 197, 2007.

[25] Pivotal Software, "Spring Statemachine," http://projects.spring.io/spring-statemachine/, 2016, accessed: 2016-02-19.

[26] H. Widmoser, "Interaction Planning for collaborative Human-Robot Assembly Tasks," Master's thesis, TU München, 2012.

[27] R. Diestel, "Graph theory," *Springer-Verlag, Heidelberg, Graduate Texts in Mathematics*, vol. 173, 2005.

[28] R. Navarro Prieto and Jose J. Cañas, "Are visual programming languages better? The role of imagery in program comprehension," *International Journal of Human-Computer Studies*, vol. 54, no. 6, pp. 799–829, 2001.

[29] D. Chang, L. Dooley, and J. E. Tuovinen, "Gestalt theory in visual screen design: a new look at an old subject," in *Proceedings of the Seventh world conference on computers in education conference on Computers in education: Australian topics-Volume 8*. Australian Computer Society, Inc., 2002, pp. 5–12.

[30] M. Schuster, C. Brand, S. G. Brunner, P. Lehner, J. Reill, S. Riedel, T. Bodenmüller, K. Bussmann, S. Büttner, A. Dömel, W. Friedl, I. Grixa, M. Hellerer, H. Hirschmüller, M. Kassecker, Z.-C. Marton, C. Nissler, F. Rueß, M. Suppa, and A. Wedler, "The LRU Rover for Autonomous Planetary Exploration and its Success in the SpaceBotCamp Challenge," in *Autonomous Robot Systems and Competitions (ICARSC), IEEE International Conference on*, 2016.

[31] F. Steinmetz and R. Weitschat, "Skill Parametrization Approaches and Skill Architecture for Human-Robot Interaction," in *Automation Science and Engineering (CASE), IEEE International Conference on*, 2016.

[4]http://www.saphari.eu/

[5]http://rmc.dlr.de/rafcon